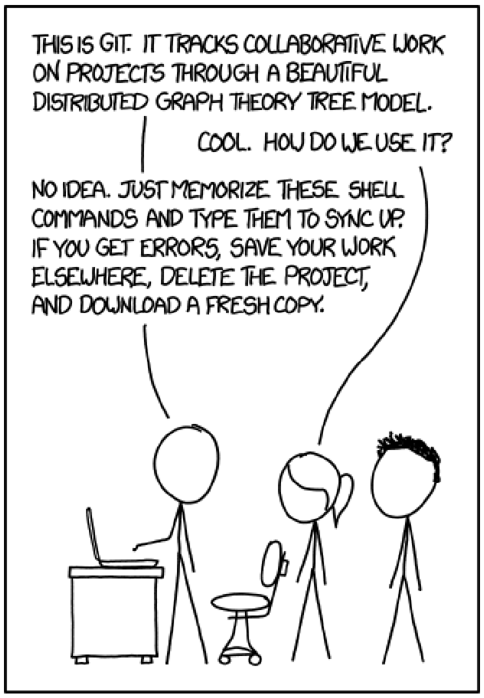
Software Engineering Processes and Management


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Workshop 6
Meet change

INTRODUCTION

The aim of this workshop is to have you thinking about configuration management, and to start working with tools for dealing with configuration management. Specifically, we will be using a *version control* tool, Git, which you will have seen in previous subjects.

Git, and other revision control tools, are not just a central place to store all of your project files for backing up; they can also be used to manage different configurations and versions of projects, and to maintain separate development branches of systems to prevent them interfering with each other.

VERSION CONTROL

Version control is one aspect of configuration management — and an important one at that. Using version control systems such as Git or Subversion is essential if one wants to maintain even one consistent version of a collaborative project. A colleague in the department tells of a small software company consisting of only two developers who did not use version control software. They allocated an entire week every two months just to merge the changes they made to their product — making them 12.5% less efficient!

SUBVERSION

In previous offerings of this subject, this workshop was written with Subversion in mind.Subversion was a once popular version control system. Refer to Appendix A for more details on tagging, branching and merging in Subversion. For this offering of this subject, we have decided to update this document to introduce Git as the version control system!

GIT

In this workshop, we will be using git. This software is readily available online and is a must have for budding software engineers!

Git is also a version control system similar to Subversion. The main difference is that Git is a *distributed version control system.* What this means is that each user has their own working copy of the repository instead of syncing to a main repository, each user can work independently before merging them together.

Subversion stores a repository in a format that looks like a file system. That is, each repository contains directories and files. Git offers many functions for modifying and using a repository, the most used being:

- Cloning to a local workspace:

  ```
  git clone https://github.com/tiow/swen90016_workshop05.git
  ```

- Adding/removing files/directories to/from the repository:

  `git add` *filename*

- Reviewing the changes made in a local workspace:

  ```
  git status
  ```

- Committing changes made in a workspace to the repository:

  `git commit` (in the directory that you want committed).

- Committing changes with a message:

  `git commit -m "message"`

- View all local branches

  `git branch`

- Branching a new version of the system, allowing multiple teams to work on separate versions at one time:

  `git branch` *branch-name* (to create the branch).

  `git checkout` *branch-name* (to start working in your new branch).

- Merging different branches back into a single branch:

  `git merge` *`branch-you-want-to-merge`*

For more information on Git, visit `https://www.atlassian.com/git`

AN EXERCISE IN CHANGE

The task in this workshop is to practice using Git.

THE SUB-SYSTEM

The sub-system is a Java-based implementation of a non-directed graph: a collection of nodes, and a collection of edges between those nodes. The graph is implemented in a class called `Graph`, which maintains a graph of integers. The source for this can be found in the `src/` directory. The interface to the class contains the following methods:

`Graph(int n)`	A constructor that creates a graph with maximum size `n`, where `n` represents the maximum number of nodes.
`addNode(int n)`	Adds a node `n` to the graph if and only if the graph is not full. If the graph is full, do nothing.
`addEdge(int m, int n)`	Add an edge (`m`, `n`) to the graph. Edges are specified by pairs of nodes. An edge be added if and only if m and n have already been added to graph as nodes.
`deleteNode(int n)`	Delete the node `n` from the graph if and only if it is not part of some edge in the graph and it exists as a node in the graph.
`deleteEdge(int m, int n)`	Delete the edge (`m`, `n`) from the graph.

The data structures used to maintain the system are:

`int [] _nodes`	An array of integers that maintains the nodes in the order that they were added.
`int _allocated`	An integer that maintains the index of the last node in `_nodes`.
`int _order`	An integer representing the maximum size of the graph.
`boolean [][] _matrix`	A two-dimensional boolean matrix that maintains the *indices* of the nodes in the graph.

The boolean matrix identifies edges in the graph. For two nodes, `m` and `n`, the edge (`m`, `n`) is in the graph if and only if, for the *indices*, `mIndex` and `nIndex` of `m` and `n` in `_nodes`, the value `_matrix[mIndex, mIndex]` is true. For example, in the following matrix, there are edges (0,2) and (1,3). The graph is undirected, so there are also edges between (2,0) and (3,1).

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \left[\begin{array}{cccc} F & F & T & F \\ F & F & F & T \\ T & F & F & F \\ F & T & F & F \end{array}\right] \end{array}$$

Therefore, to check whether an edge exists, one must first *lookup* where the two nodes sit in the array `_nodes`, and then test whether these two indices map to true in the matrix. There exists a protected method `_lookup` look up the index of a node.

In the directory `test`, there is an automated test driver (`Driver.java`), which tests the functionality of each method.

To compile the source, use '`make compile`'. To run the tests, use '`make test`'.

THE PROBLEMS

Your group has been assigned responsibility of modifying the source code and tests of the graph sub-system. Management has identified two faults that must be fixed, and two new features that must be added. The current test script does not reveal the faults, so it must be updated to test this. In addition, the test script needs to be updated to test and use the new features.

The following changes need to be made to the graph program and its test script:

1. There is a fault in the `deleteNode` method. In the test driver that exists already, some tests for `deleteNode` exist — a node that is part of an edge. However, coincidentally, the test passes.

   Two people will be assigned to this change: one must fix the fault, the other must add a test case that detects the fault.

2. There is a fault in the `addNode` method. Again, the test script fails to detect this fault.

Two people will be assigned to this change: one must fix the fault, the other must add a test case that detects the fault.

3. Two new methods must be added: one to check the whether a node is present in the graph, and one to check whether an edge is present in the graph:

`isNode(int n)`	Returns true if and only if the node `n` is in the graph.
`isEdge(int m, int n)`	Returns true if and only if the edge (`m`, `n`) is in the graph.

   Four people will be assigned to this change: one to add `isNode`, one to add tests for `isNode`, one to add `isEdge`, and one to add tests test for `isEdge`.

Each student will be handed an instruction sheet telling them which changes they have to make, and how to make them.

YOUR TASKS

Your tasks for the workshop are:

Checkout/download and compile (5 mins). The source code and test driver are available on Github

```
git clone https://github.com/tiow/swen90016_workshop05.git
```

Plan (10 mins). Discuss how you are going to implement the plan given to you by management.

Do (20 mins). Implement the plan, uploading the source code back to your own Github servers.

Discuss (10 mins). Compare the experiences between the two teams. How was the task coordinated? Was there any issues that arose? Were you able to complete the task in time?

APPENDIX A

TAGGING, BRANCHING, AND MERGING (SUBVERSION) [1]

In this section, we explore a few of the more advanced functions of version control systems, from the view of Subversion.

Tagging. Tagging is the simpler of the two concepts so we'll start there. Tagging is simple assigning a tag – a name – to a snapshot of your repository. In Subversion, this is really easy to understand as Subversion uses global version numbers for files, so every version is a automatically "tagged" with a unique number. In CVS tagging has more appeal as each file has its own version so keeping track of which version aligns to what is difficult if it's not managed.

Now, the reason one would want to use tagging is to assign names to important and/or significant milestones in the development of your source files. Early on this might be a "finalised", reviewed and signed off version of your requirements specification, or a version of your architecture that your team is happy to proceed with. Down the track, this will most likely be used for source files when your product hits an initial beta version or a release, such as "1.0", that will be shipped to the client.

That's pretty much all there is to tagging, the only thing that you have to keep in mind when working in Subversion (and probably some other version control systems) is that tagging is handled internally in the same way as branching, which I will discuss now, and the only difference is one of pure semantics.

Branching and merging. If you think of your repository development as occurring in a straight line (called a "mainline" or "trunk") then a branch is just what you would expect: an offshoot of your main development that can continue independently. Two important things to understand with branching are that only files which are changed in that branch are actually re-versioned and that any changes you make to a branch can be merged back in to the mainline at a later time. Perhaps the best way to explain this is with an example.

Say you have just reached version 1.0 of your product and you ship this version out but plan on continuing development. Making a branch of your 1.0 code is good practice because it means you'll always be able to easily restore to that point and compile and ship another copy if need be. This, of course, is the same as tagging. Branching will allow you to go one step further - you can continue development on the 1.0 branch. In three weeks time a customer alerts you to a bug in your 1.0 product. It is obviously not ideal to try and replicate the 1.0 code from your new "2.0" code, nor is it a good idea to try and ship "2.0" early to fix this lone bug. With your 1.0 branch, however, this is trivial - you still have an exact copy of the code that was shipped out to the customers and you can release a new 1.1 version with the bug fix. Note that you now have a "2.0" mainline, which is where all your new code lives, and a "1.0" branch of your product that contains the old version with the applied bug fix.

Now, another common use for branching (at least in Subversion) is to allow each developer, or development team, to work on their own copy of the code. This allows for a minimal number of conflicts for all the constant updates and commits that you'll want to do and is most useful when

[1] Courtesy of Chris Norton.

a large number of disruptions are expected, such as those that occur when rewriting a core part of a library or adding in a major new feature. When a developer is finished on their features, fixes or whatever they can merge their code back into the mainline (or another branch).

Merging can be thought of as the opposite of branching. It's the process of combining files from separate branches back together again. This might result in the effective elimination of a branch or it might just be a way of synchronising some files that are common between branches. In the example given before, a bug fix that is applied to a file from the 1.0 branch (that has not seen significant modifications for the 2.0 branch) is a perfect candidate for merging back into the mainline.